UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

DESARROLLO DE UNA PLATAFORMA HIL PARA UN CONTROLADOR DE VUELO.

Enrique Cabrerizo Fernández
Tutor: Himar Alonso Díaz
Ponente: Eloy Anguiano Rey

Junio 2016

UNIVERSIDAD AUTÓNOMA DE MADRID
Escuela Politécnica Superior
Francisco Tomás y Valiente, nº 11
Madrid, 28049
España

Desarrollo de una plataforma HIL para un controlador de vuelo.
Enrique Cabrerizo Fernández, Junio 2016

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Desarrollo de una plataforma HIL para un controlador de vuelo.

AUTOR: Enrique Cabrerizo Fernández
TUTOR: Himar Alonso Díaz
PONENTE: Eloy Anguiano Rey

Escuela Politécnica Superior
Universidad Autónoma de Madrid

Junio 2016

AGRADECIMIENTOS

Al equipo de UAV Navigation y en particular a mi tutor Himar, por confiar en mí para el desarrollo de este proyecto y enseñarme algo nuevo cada día.

A la gente de mi clase, por ser uno de los mejores grupos que he tenido la suerte de conocer y en especial a Rual, mi amigo y compañero de interminables prácticas, por ayudarme siempre y animarme en los momentos más difíciles. Estos años habrían sido mucho más duros sin vosotros.

A mis amigos “de toda la vida”, por ser mi segunda familia y no haberme echado todavía de casa.

A mi familia, tanto a los que están como a los que ya se fueron, por su amor incondicional.

Resumen

Actualmente, la cantidad de sistemas embebidos presentes en todo tipo de ámbitos se encuentra en continuo crecimiento. Algunos de estos ámbitos, como el sector aeronáutico, son entornos críticos en los que un error en el sistema puede tener altos costes, tanto económicos como humanos. Así, se hace necesaria la verificación del correcto funcionamiento de estos sistemas previa a su integración en el entorno final. Esta labor puede ser muy compleja si no se adopta el enfoque adecuado. Una de las posibles aproximaciones es la simulación *Hardware-in-the-Loop* (HIL). Una simulación HIL trata de emular el comportamiento y las características del entorno real en el que se integrará el sistema para obtener información precisa acerca de su comportamiento.

En este proyecto se ha diseñado e implementado una plataforma HIL para la unidad de control de vuelo VECTOR con el objetivo de facilitar la detección de errores en el software de piloto automático y el diseño de casos de prueba.

El resultado ha sido una implementación que hace uso de diversas tecnologías interconectadas entre sí (FPGA, simulador de vuelo *X-Plane*, *Teensy* y aplicación *SimAp*) para simular los sensores de un vehículo aéreo no tripulado (*Unmanned Aerial Vehicle*, UAV) y procesar las respuestas de la unidad de control de vuelo. La arquitectura de la aplicación ha sido diseñada de forma modular con el fin de facilitar la posterior creación de nuevos módulos y funcionalidades que permitan aumentar el realismo de la simulación.

Palabras clave: HIL, UAV, sistema embebido, *Teensy*, FPGA, *X-Plane*, VECTOR

Abstract

Nowadays, the amount of embedded systems found in every kind of area is constantly growing. Some of these, such as the aeronautical sector, are critical environments in which any error in an embedded system may come at high costs, both financial and human. Therefore, thorough testing and validation of these systems prior to their integration into the target environment is mandatory. Such tasks might be difficult to fulfill if the right approach is not taken. One of the possible ways to face this issue is Hardware-in-the-Loop (HIL) simulation. HIL simulation aims to emulate the behaviour and features of the target environment in order to get accurate information about the system being tested.

In this project, a HIL platform for the flight control unit VECTOR has been designed and developed in order to make autopilot software errors detection and test case designing easier.

The outcome of this project is an implementation that uses several interconnected technologies (FPGA, *X-Plane* flight simulator, *Teensy* and *SimAp* application) to simulate the sensors of an unmanned aerial vehicle (UAV) and handle the responses of the flight control unit. The application architecture has been designed in a modular way in order to ease the subsequent creation of new modules and functionalities that increase the realism of the simulation.

Keywords: HIL, UAV, embedded system, Teensy, FPGA, X-Plane, VECTOR

Tabla de contenidos

LISTAS

Lista de figuras

Lista de tablas

GLOSARIO

acelerómetro
Dispositivo utilizado para medir aceleraciones en un cuerpo.

condición de carrera
"Comportamiento anómalo de un sistema causado por la dependencia crítica en la temporización relativa de los eventos" [8].

dataref
Variable que interviene en la fase simulación de vuelo de *X-Plane*.

giróscopo
Dispositivo utilizado para medir la orientación de un cuerpo.

magnetómetro
Dispositivo utilizado para medir la magnitud y orientación de un campo magnético.

planta real
Sistema real en el cual se integra la UUT.

plug-and-play
"Hardware o software que, tras ser instalado (*plugged in*), puede ser inmediatamente usado (*played with*), en contraposición al hardware o software que requiere configuración." [8].

plugin
Aplicación que extiende o aporta una funcionalidad nueva a otra aplicación sobre la que se ejecuta.

POLAR
Unidad de medición inercial (IMU, *Inertial Measurement Unit*) integrada en la unidad VECTOR.

recolector de basura
Mecanismo de gestión de memoria implementado en algunos lenguajes de programación que se encarga de la reserva y liberación de memoria de forma transparente al programador.

simulación de planta
Conjunto de Hardware y Software de la plataforma HIL que interactúan con la UUT simulando el comportamiento de la planta real de la forma más precisa posible.

sistema embebido
Combinación de hardware y software desarrollado para realizar una función específica. Se suele encontrar integrado en un sistema mayor con restricciones de funcionamiento en tiempo real.

telemetría

Sistema de medición de magnitudes físicas que permite transmitir los datos obtenidos a un observador lejano.

VECTOR

Sistema de piloto automático desarrollado por la empresa UAV-Navigation. A lo largo de este TFG, esta unidad será referida a menudo como UUT.

ACRÓNIMOS

ASIC *Application-Specific Integrated Circuit*.

CPU *Central Processing Unit*.
CRC *Cyclic Redundancy Check*.
CS *Chip Select*.

FPGA *Field Programmable Gate Array*.

GNSS *Global Navigation Satellite System*.
GPL *General Public License*.
GPS *Global Positioning System*.

HID *Human Interface Device*.
HIL *Hardware-in-the-Loop*.

ICD *Interface Control Document*.
IDE *Integrated Development Environment*.
IMU *Inertial Measurement Unit*.
IPv4 *Internet Protocol version 4*.

LGPL *Lesser General Public License*.

MISO *Master Input Slave Output*.
MOC *Meta-Object Compiler*.
MOSI *Master Output Slave Input*.

PLL *Phased-Lock Loop*.
PWM *Pulse-Width Modulation*.

RPM *Revolutions Per Minute*.
RS232 *Recommended Standard 232*.

SPI *Serial Peripheral Interface*.

TCP *Transmission Control Protocol*.
TSIP *Trimble Standard Interface Protocol*.

UAV *Unmanned Aerial Vehicle*.
UIC *User Interface Compiler*.
USB *Universal Serial Bus*.

UUT *Unit Under Test*.

XML *eXtensible Markup Language*.

1 Introducción

La simulación *Hardware-in-the-Loop* (HIL) es una técnica utilizada para el desarrollo y validación de un sistema embebido, al que en adelante nos referiremos como *Unit Under Test* (UUT). En la simulación HIL, la UUT se conecta a la simulación de planta e interactúa con ella de la misma forma que lo haría con la planta real. Para que la simulación HIL sea fiable, la UUT debe realizar exactamente las mismas tareas que realizaría en la planta real, por lo tanto, la simulación HIL no debe introducir ni eliminar carga de trabajo sobre la UUT, de lo contrario se desvirtuaría su funcionamiento y el resultado de la simulación podría diferir del esperado en una situación real.

1.1. Antecedentes y Motivación

Este TFG surge de la necesidad de validar las unidades de control de vuelo VECTOR, desarrolladas por la empresa UAV Navigation [7], con la finalidad de probar su fiabilidad y facilitar la detección de posibles errores en el software de piloto automático.

Hasta la fecha actual, las pruebas realizadas sobre dichas unidades presentan dos formatos claramente diferenciados:

- Pruebas software: La UUT se carga con un software de simulación que emula una unidad en vuelo. Dicha unidad ejecuta tanto la simulación, como la respuesta.
- Pruebas en planta real: Se equipa un UAV (*Unmanned Aerial Vehicle*) con la unidad de control de vuelo, se establece un plan de vuelo en dicha unidad y se observa el comportamiento del UAV, obteniendo datos de telemetría en una estación de tierra para su posterior análisis. En caso de fallo del sistema, el control se pasa a modo manual y el UAV pasa a ser controlado por un piloto de *Unmanned Aerial Vehicle*s (UAVs) certificado.

Estas pruebas presentan algunos inconvenientes:

- La UUT utiliza un sistema operativo en tiempo real. La simulación por software de las condiciones de vuelo, modifica la ejecución normal del piloto automático introduciendo una gran cantidad de cálculos que en condiciones de vuelo no se van a realizar. Así, es posible que la funcionalidad se vea alterada, provocando retrasos en la temporización del sistema operativo, llegando incluso a omitir ciertas tareas de inferior prioridad en

cada ejecución del bucle principal. Además, ciertas condiciones de carrera que se darían en el sistema, podrían no producirse de esta forma.

- En las pruebas en planta real, tras un fallo del sistema, se requiere de la pericia del piloto para retomar el control y estabilizar el UAV. Aun así, no siempre es posible evitar accidentes y que se produzcan daños en los equipos, con el consiguiente coste de reparación.
- Los daños materiales, no sólo repercuten económicamente por el coste de reparación. La indisponibilidad del equipo ralentiza la corrección de errores ante la imposibilidad de probar una posible solución, paralizando el desarrollo del producto.
- Además, las pruebas en planta real requieren una preparación previa que consume una gran cantidad de tiempo (desplazamientos con el equipo) y puede estar limitada según las condiciones temporales y climatológicas por la normativa vigente [13].

Todos estos problemas pueden evitarse con el desarrollo de una plataforma HIL. Si bien en última instancia se hace necesaria la validación del sistema en planta real, la simulación HIL aporta el siguiente valor añadido:

- Posibilidad de simular condiciones climatológicas extremas así como errores en el UAV, tales como fallo de motor, tren de aterrizaje, etc.
- Facilidad para la reproducción de errores de software en el piloto automático, al tener el control sobre todos los elementos que intervienen en el bucle.

1.2. Alcance y Objetivos

El objetivo principal del proyecto es realizar una prueba de concepto para verificar la viabilidad y utilidad de la plataforma HIL de cara a la detección de errores y a la mejora en la respuesta del software de control de vuelo incluido en la unidad VECTOR, mediante la realización de un prototipo simplificado. Además, de cara al posible éxito de la plataforma, se debe plantear una solución que sea fácilmente portable y ampliable.

En este proyecto se presentará una primera aproximación a la plataforma HIL y su objetivo fundamental será proveer a la UUT con un *input* similar en formato y temporización al que recibiría en vuelo e interpretar el *output* de la unidad de forma similar a como lo haría el UAV al que estaría conectado para producir un nuevo *input* que realimenta el bucle. En la Figura 1.1 se presenta un esquema general de una plataforma HIL

Este proyecto no abarca la detección de errores *hardware* en las unidades, ya que existen pruebas específicamente diseñadas para ello. Tampoco pretende corregir los fallos *software* en las unidades, únicamente facilitar su detección.

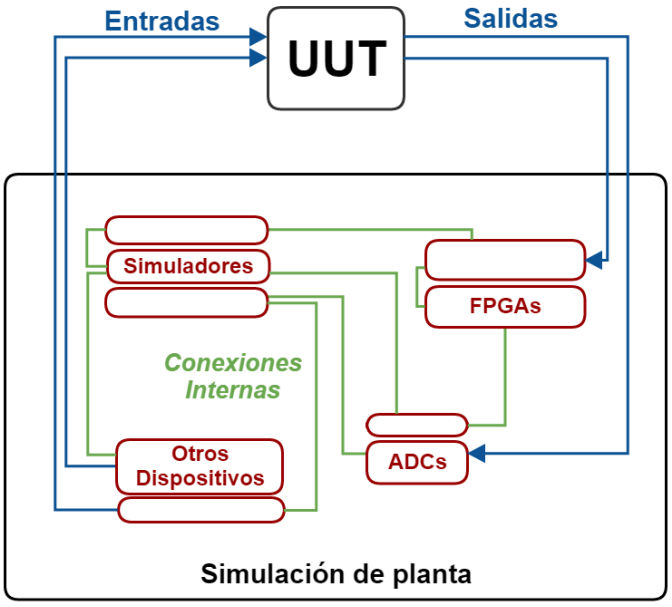


Figura 1.1: Esquema general de una plataforma HIL.

1.3. Estructura del documento

A lo largo del capítulo 2 se analizarán las diferentes tecnologías que se emplearán en el desarrollo de la plataforma. Se describirán brevemente las características de los dispositivos FPGA, *Teensy* y VECTOR (UUT) utilizados. Se comentarán las características principales del simulador de vuelo *X-Plane* [10] y del *plugin ExtPlugin* [5] empleado. También se introducirá la biblioteca multiplataforma Qt poniendo especial énfasis en las características principales que han sido utilizadas en este proyecto y se concluirá con una descripción de los protocolos utilizados para la comunicación entre los distintos componentes de la plataforma HIL.

En el capítulo 3 se presentará el diseño de la plataforma HIL y se detallará el desarrollo y modo de empleo de cada una de las tecnologías descritas en el capítulo 2 para la consecución de los objetivos del proyecto.

En el capítulo 4 se describirán las pruebas realizadas a la plataforma en general y sus componentes en particular, en los casos que proceda.

Finalmente, en el capítulo 5 se presentarán las conclusiones a las que se ha llegado y se aportarán algunas ideas para la mejora y la extensión de la plataforma HIL en trabajos futuros.

2 Estudio de las tecnologías a utilizar

A lo largo de este capítulo se describen todos los elementos que intervienen en la plataforma HIL, haciendo especial hincapié en las características útiles para el proyecto. Se comenzará describiendo el dispositivo VECTOR, principal objeto de test de la plataforma, y la IMU que integra (POLAR) y se presentarán sus características principales. A continuación se comentarán brevemente las características y propósito de los dispositivos FPGA y *Teensy* empleados en la primera fase del bucle HIL.

Posteriormente se presentan el simulador de vuelo *X-Plane* y el *plugin ExtPlugin* y se comentará su utilidad y forma de uso.

A continuación se presentará la biblioteca *Qt*, se describirán las partes que componen el entorno de desarrollo *QtCreator* y los mecanismos de la biblioteca que han sido utilizados en el desarrollo de este TFG, así como el mecanismo de *signals* y *slots*, manejo de hilos, etc.

Para finalizar, se describirán los siguientes protocolos y estándares empleados para conectar y comunicar las distintas partes de la plataforma: TSIP, HID y SPI.

2.1. VECTOR

La unidad de control de vuelo VECTOR, desarrollada por la empresa UAV Navigation [7], es un dispositivo de piloto automático para UAVs capaz de controlar todas las fases de vuelo (despegue, navegación y aterrizaje). Algunas de sus características principales son:

- Dos CPUs redundantes con capacidad para resetearse en pleno vuelo.
- IMU integrada, de forma que el VECTOR no necesita sensores externos para controlar completamente un UAV. La IMU que integra también ha sido desarrollada por la empresa UAV Navigation [7] y se comercializa bajo el nombre POLAR.
- Sistema operativo en tiempo real *FreeRTOS*.
- 24 entradas/salidas configurables (discretas, PWM, RPM *count*...).
- Puertos de comunicación: CAN, Ethernet, RS232.
- 7 entradas analógicas.

Figura 2.1: VECTOR

En este TFG no se estudiará en detalle el funcionamiento de la unidad, lo único que se precisa saber es que en cada ciclo de la ejecución del bucle principal del piloto automático, los valores de las señales de cada una de las 24 salidas de la unidad, dependen directamente de los datos proporcionados a la unidad por la IMU integrada. En una prueba en planta real, estas salidas se conectan a los servomotores que mueven las superficies de vuelo del UAV, permitiendo a la unidad VECTOR controlar el rumbo de acuerdo al plan de vuelo establecido.

2.2. IMU (POLAR)

Una Unidad de Movimiento Inercial (*Inertial Measurement Unit*, IMU) es un dispositivo electrónico, que combina la información obtenida por medio de acelerómetros, giróscopos y magnetómetros para indicar la variación en la velocidad, angulación y fuerzas a las que está sometido un cuerpo.

En el caso de la unidad POLAR, estos datos se combinan, además, con información GNSS (el GNSS más utilizado es el conocido como GPS) para realizar una estimación precisa de la posición, orientación espacial, velocidad, aceleración y toda una serie de variables útiles para la navegación. Estos datos se formatean de acuerdo al protocolo TSIP (descrito en la sección 2.9) y se envían al VECTOR por una interfaz RS232 con una temporización fija. El formato y la temporización de dichos paquetes se detallarán en la sección 3.3.4.

2.3. FPGA

Una FPGA es una matriz de elementos lógicos programables (y generalmente reprogramables) mediante lenguajes de descripción hardware (VHDL, Verilog...) para conseguir una funcionalidad específica. Se distinguen de los ASICs precisamente en su capacidad para ser reprogramados aunque generalmente son más lentos y tienen un mayor consumo de potencia que éstos.

Figura 2.2: POLAR

La utilidad de la FPGA en este proyecto es recibir las salidas del dispositivo VECTOR y procesarlas en paralelo (con recursos hardware dedicados) para construir un paquete de información que agrupa los valores de todas las salidas y enviarlo por una interfaz SPI. El formato del paquete se describe en la sección 3.2.

La FPGA utilizada pertenece a la familia *Cyclone* del fabricante *Altera*. Esta familia destaca por su buena relación calidad precio para desarrollos que no requieran demasiados recursos hardware ni operaciones a alta velocidad (del orden de decenas o incluso pocas centenas de MHz).

La FPGA se encuentra integrada en la placa de evaluación *DE0-Nano* fabricada por *Terasic* [3] y cuenta con las siguientes características útiles para el proyecto:

- Cristal de cuarzo en placa a 50MHz.
- 72 pines de entrada/salida.
- Precio económico ($61,00).

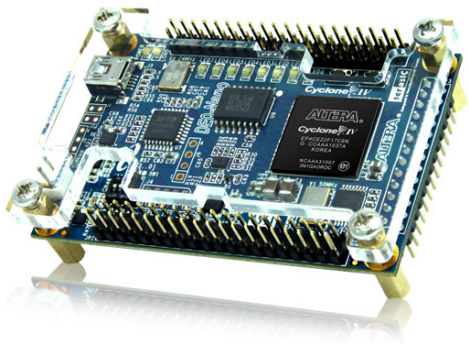


Figura 2.3: Placa de desarrollo *DE0-Nano* con FPGA *Cyclone*

La placa ha sido proporcionada con la funcionalidad ya implementada por la empresa UAV Navigation [7] por lo que, en adelante, se considerará el conjunto VECTOR+FPGA como un único elemento que acepta entradas con un formato específico y proporciona paquetes de información en respuesta por un canal SPI.

2.4. *Teensy* 3.2

El dispositivo *Teensy* es un microcontrolador programable por USB que cuenta con las siguientes características:

- Procesador ARM CortexM4 de 32 *bits* a 72MHz.
- 64Kb de memoria RAM y 256Kb de memoria flash.
- Hasta 34 entradas/salidas digitales y 21 entradas analógicas.
- Comunicación USB, SPI y Serie, entre otros.
- Capacidad para emular cualquier dispositivo USB (teclados , ratones, *joysticks*, etc.).
- Precio económico ($19,80)

Teensy es compatible con una gran cantidad de bibliotecas disponibles para *Arduino* [14] y se programa utilizando el IDE de *Arduino* [14], cargado con el *plugin Teensyduino*. La lista completa de librerías *Arduino* compatibles con *Teensy* y la descarga del *plugin Teensyduino* se pueden encontrar en la web oficial [9].

Una de las funcionalidades del dispositivo utilizadas durante este proyecto, es la comunicación por SPI. Para ello se ha empleado la biblioteca *spi4teensy* [11], avalada por los creadores del dispositivo como una opción más optimizada que la librería SPI de *Arduino* por defecto.

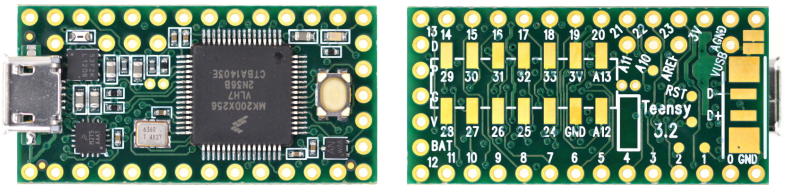


Figura 2.4: Vista frontal y trasera del dispositivo *Teensy*

2.5. *X-Plane* y *plugin ExtPlane*

El programa *X-Plane* [10] es un simulador de vuelo multiplataforma que se basa en la geometría de la aeronave y otros parámetros físicos como el peso, centro de masa, etc, para realizar una simulación precisa de su comportamiento en vuelo. Algunas de sus características principales son las siguientes:

- Simulación realista de condiciones atmosféricas.
- Simulación de fallos en los sistemas de la aeronave.
- Capacidad para crear y personalizar aeronaves mediante la herramienta *PlaneMaker*.
- Soporte para *joysticks*.

- Soporte para *plugins*.
- Facilidad para acceder a todas las variables que intervienen en la simulación de vuelo (*datarefs*), por ejemplo: vector velocidad de la aeronave, vector de aceleración, orientación, etc. También es posible definir *datarefs* propios.

El *plugin* utilizado, *ExtPlane* [5], permite acceder y modificar *datarefs* mediante *sockets* TCP. Algunas de sus características son:

- Capacidad para soportar múltiples conexiones concurrentes.
- Soporte para diferentes tipos de *datarefs*, entre ellos, los tipos *float* y *double*.
- Posibilidad de seleccionar la tasa de actualización de acuerdo a la precisión. De forma que solamente se envía un valor de *dataref* si ha cambiado por encima de un cierto umbral configurable.

En este proyecto, se usará el *plugin* únicamente para leer valores remotamente, aunque también existe la opción de modificarlos. Su modo de empleo es muy sencillo y se resume en los siguientes pasos:

- Establecer conexión TCP.
- Suscribirse a un *dataref* (o varios) enviando comandos de texto con formato:

  ```
  sub {dataref_name} [accuracy]
  ```

  Donde *[accuracy]* es un parámetro numérico opcional que indica el valor mínimo que debe variar un *dataref* respecto al último valor notificado para que se envíe de nuevo. Este parámetro es muy útil para no saturar la comunicación cuando se suscribe un gran número de *datarefs*.
- Procesar la respuesta obtenida, que tiene el siguiente formato de texto:

  ```
  u{t} {dataref_name} value
  ```

  Donde {t} indica el tipo, siendo 'f' para float y 'd' para double.

Existen más de 3500 *datarefs* oficiales [1] pero en este proyecto se utilizarán inicialmente menos de 30.

2.6. Biblioteca *Qt*

Qt [6] es una biblioteca multiplataforma para el desarrollo de aplicaciones con interfaz gráfica de usuario en lenguaje C++ desarrollada originalmente por la empresa *Trolltech*, que fue adquirida posteriormente por *Nokia*. Actualmente *Qt* se ofrece en dos licencias, una comercial y una pública bajo una combinación de licencias GPL y LGPL, dependiendo de la versión y el módulo de *Qt*.

Algunas plataformas soportadas oficialmente por *Qt* [6] son *X11* (*GNU/Linux*, *FreeBSD*, *Solaris*...), *Windows* (XP, 7, 8, 10...), *OS X*, *iOS*, *Android* y algunas distribuciones de *Linux* embebido.

A continuación se detallará el funcionamiento y las características particulares de los aspectos de la biblioteca utilizados en el proyecto.

2.6.1. *QtCreator*

QtCreator es el IDE oficial de *Qt*. Está disponible para descarga gratuita en la web [6] e incluye los siguientes elementos:

- *QtDesigner*: Herramienta para la creación de interfaces de usuario de forma sencilla, pinchando y arrastrando elementos de la interfaz como botones, cuadros de texto, etc (*widgets*). El resultado del diseño es un fichero '.ui' con contenido XML.
- Compilador de interfaz de usuario (*User Interface Compiler*, UIC): Herramienta que traduce el contenido del fichero '.ui' a código C++.
- Compilador de Meta-Objetos (*Meta-Object Compiler*, MOC): Herramienta que se encarga de implementar y dotar a los objetos de las extensiones en funcionalidad propias de *Qt*. La herramienta recorre los archivos de cabecera del proyecto buscando la macro Q_OBJECT. Cuando encuentra una coincidencia, crea un nuevo archivo fuente y lo vincula a la clase anterior.
- qmake: Herramienta para la generación automática de ficheros *Makefile*. El *Makefile* se genera a partir de la información contenida en un fichero de configuración que permite añadir de forma sencilla las librerías que se quieran incluir en el proyecto. Esta herramienta incluye llamadas automáticas al compilador de Meta-Objetos y al compilador de interfaz de usuario.

Además, *QtCreator* incluye elementos típicos de los IDE como capacidad para depurar y herramientas para configurar el despliegue de aplicaciones.

2.6.2. La clase *QObject*

La clase básica en *Qt* es la clase *QObject*. Una típica declaración de clase en *Qt* comienza de la siguiente manera:

```
class ExampleClass : public QObject
{
        Q_OBJECT

        ExampleClass(QObject *parent = 0);
        ~ExampleClass();

        // (...)
}
```

La clase *QObject* proporciona la base para que un objeto pueda implementar todas las funcionalidades extra añadidas por el MOC. La macro Q_OBJECT se usa para indicar al MOC que active dichas funcionalidades en la nueva clase.

El constructor de la clase puede recibir opcionalmente un objeto *parent*. Esta es una de las funcionalidades añadidas por el MOC y permite que el objeto sea automáticamente destruido cuando se destruye el objeto *parent*, simplificando en gran medida la gestión de memoria, ya que C++ no cuenta con un recolector de basura y toda la liberación de recursos depende del programador.

Únicamente es necesario que una clase extienda *QObject* si se quiere hacer uso de las funcionalidades aportadas por el MOC.

2.6.3. Mecanismo de *signals* y *slots*

Otra funcionalidad añadida por el MOC es el mecanismo de *signals* y *slots*. Tiene el objetivo de facilitar la comunicación entre objetos y en particular en este proyecto se utilizará para la comunicación entre hilos.

Continuando con el ejemplo iniciado anteriormente, la declaración de la clase podría continuar:

```
{
        // (...)

signal:
        void signal1();
        void signal2(int, float);
public slot:
        float slot1(int);
        float slot2(float, int);
        void doSomething();
}; //end
```

Para establecer una conexión entre una señal y un *slot* se invoca la siguiente función (suponiendo que *emitter* y *receiver* son objetos de tipo *ExampleClass*):

```
connect(emitter, SIGNAL(signal2), receiver, SLOT(slot1));
```

Cada vez que se ejecute la sentencia `emit signal2(intValue, floatValue)` desde el objeto *emitter*, se invocará el método `slot1(intValue)` sobre el objeto *receiver*.

El número de argumentos que envía una señal debe ser mayor o igual que el número de argumentos que recibe el *slot*. Todos los argumentos de más se ignoran, pero los que no se ignoran deben coincidir en el tipo. Por ejemplo, no sería posible conectar *signal1* a *slot1* por falta de argumentos, ni *signal2* a *slot2*, por discrepancia de tipos.

En una ejecución de un *slot*, provocada por una señal, se ignora el valor de retorno. No

obstante los *slots* son métodos que pueden ser invocados manualmente y por esa razón pueden tener un valor de retorno distinto de *void*.

2.6.4. *QThread*

La forma general de trabajar con hilos en *Qt* es haciendo uso de la clase *QThread*. El siguiente código muestra la forma más simple de crear y ejecutar un hilo en *Qt*:

```
QThread *th1 = new QThread();
ExampleClass *task = new Task();
task->moveToThread(th1);
connect( thread, SIGNAL(started()), task, SLOT(doSomething()));
thread->start();
```

La llamada `moveToThread(th1)` de la clase *QObject* cambia la *thread affinity* del objeto *task* al hilo *th1*. Esto quiere decir que todas las señales y eventos que reciba el objeto, producirán una respuesta (ejecución del *slot* o del manejador de eventos asociado) que siempre se ejecutará en el hilo *th1*. Esto no impide que se puedan ejecutar métodos de dicho objeto en otro hilo, como ocurriría si después del código anterior se ejecuta `task->doSomething()`. Ese tipo de prácticas se deben evitar ya que de no controlar correctamente la casuística, se podrían dar condiciones de carrera que llevasen el objeto a un estado inconsistente. El tipo de funciones que realiza adecuadamente este control se dice que son *thread-safe*.

En el ejemplo anterior se puede ver como se asocia la señal *started* del hilo *th1* al *slot doSomething* del objeto *task*, de tal forma que cuando se inicia el hilo se emite la señal (comportamiento por defecto), provocando que el *slot* se ejecute en dicho hilo.

2.7. Protocolo HID

El protocolo HID (*Human Interface Device*) nace con la creación del estándar USB y a consecuencia de la versatilidad de éste. El objetivo del estándar USB era el de unificar la forma de conectar los periféricos a los computadores y hacerlo de forma *plug-and-play*. Como la cantidad y funcionalidad de dispositivos disponibles es muy amplia, se hace necesaria una manera de distinguir las particularidades de cada dispositivo. En una primera instancia, los dispositivos se dividen en clases [2] que sirven para identificar el *driver* con el que debe ser manejado el dispositivo en el lado del computador. Una de estas clases corresponde a dispositivos genéricos HID.

En una segunda instancia, el dispositivo proporciona información al computador sobre el tamaño y formato de los datos que va a enviar mediante un *array* de *bytes* conocido como *Device Descriptor* [4]. El computador debe analizar el *array* e interpretar el formato antes de poder comunicarse con el dispositivo. Finalmente el dispositivo puede empezar a enviar los datos con el formato indicado para que sean interpretados por el computador.

En este proyecto se ha modificado la librería USB del dispositivo *Teensy* para personalizar el formato del paquete de *joystick* por defecto.

2.8. Protocolo TSIP

Originalmente el protocolo TSIP se creó para comunicar dispositivos con sensores GPS con el objetivo de enviar comandos y modificar su configuración. Se basa en la transmisión de paquetes de información con las siguientes características:

- Se definen dos caracteres especiales: DLE (0x10) y ETX (0x03).
- Los paquetes comienzan con el caracter DLE.
- Cuando el caracter DLE sea parte del contenido de un paquete, se incluye otro caracter DLE precediendo al anterior.
- Los paquetes son siempre de un tamaño menor o igual a 256 *bytes*.
- Los paquetes terminan con la secuencia DLE, ETX.

El dispositivo POLAR, integrado en el VECTOR, se comunica con el resto de elementos mediante una variante de este protocolo y fija las características del canal de comunicación. Las particularidades de esta variante TSIP son las siguientes:

- El orden de los *bytes* es *Little Endian*.
- Los paquetes tienen identificadores de dos *bytes* (a diferencia del protocolo original, que tenía solamente uno).
- Los últimos 4 *bytes* del paquete están formados por un CRC de 32 *bits* del paquete completo (incluyendo el identificador).

En la figura 2.5 se presentan los pasos para la creación de un paquete TSIP válido.

2.9. Estándar de comunicaciones SPI

El bus de comunicaciones SPI se utiliza para comunicar dispositivos en serie de forma síncrona, basado en la jerarquía *maestro-esclavo*.

En una comunicación por SPI, el dispositivo *maestro* y los dispositivos *esclavos* disponen de los siguientes pines de entrada/salida:

- MISO: Canal por el que el *maestro* recibe datos y los *esclavos* envían. Llamado DIN en la placa *Teensy* (véase el Apéndice A).
- MOSI: Canal por el que el *maestro* envía datos y los *esclavos* reciben. Llamado DOUT en la placa *Teensy* (véase el Apéndice A).
- SCK: Señal de reloj para sincronizar los dispositivos *maestro* y *esclavo* conectados. El dispositivo maestro es el encargado de generar esta señal.

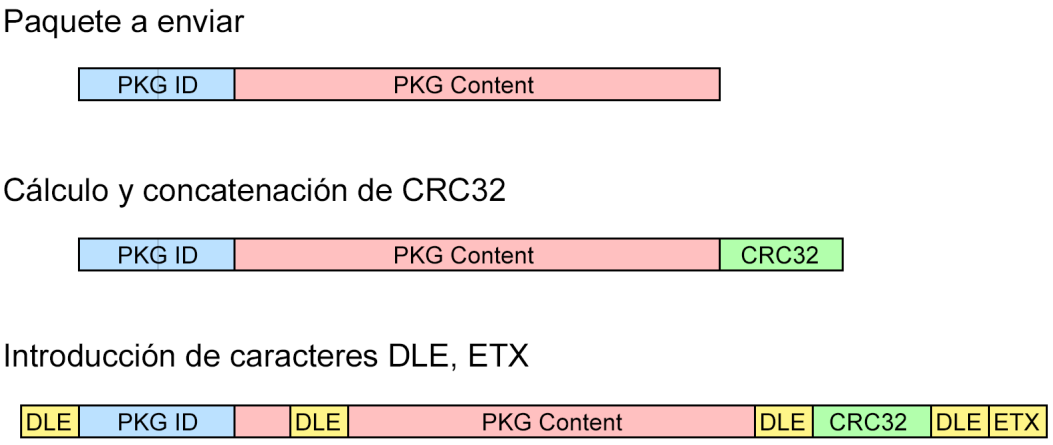


Figura 2.5: Pasos para la construcción de un paquete TSIP

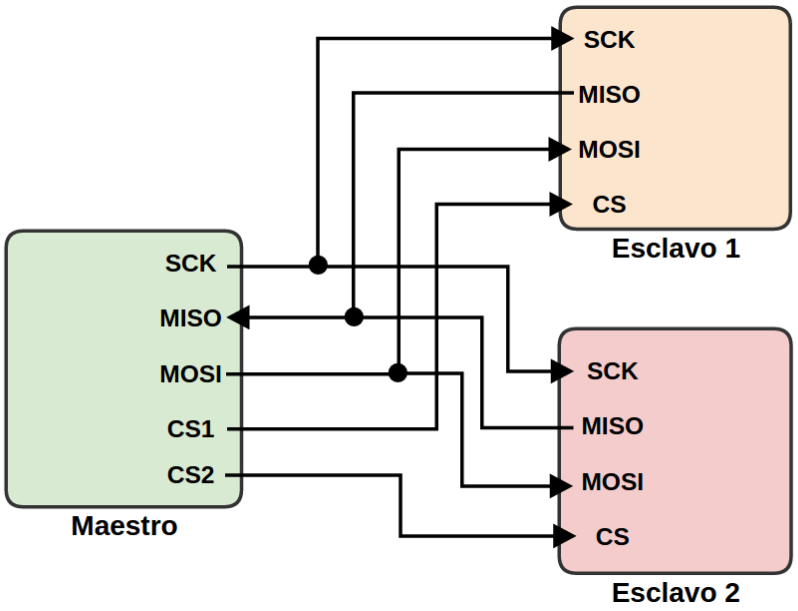


Figura 2.6: Diagrama de conexiones SPI entre un dispositivo *maestro* y dos *esclavos*

- CS: Señal que selecciona el dispositivo *esclavo* con el que iniciar la comunicación. El dispositivo *maestro* debe tener al menos tantos pines CS como dispositivos esclavos quiera conectar.

Para hacer posible la comunicación se establecen unos *bits* de configuración en cada dispositivo:

- *CPOL*: Polaridad del reloj. Indica si el estado por defecto en espera del reloj es a nivel alto o a nivel bajo.
- *CPHA*: Fase de la captura de datos. Indica si la captura del dato disponible en los pines MISO/MOSI se debe hacer en los flancos de reloj pares (considerando 0 el primer flanco) o impares.
- *DORD*: Orden de *bit*. Indica si se envía primero el *bit* más significativo (valor 0), o el *bit* menos significativo (valor 1).

En este proyecto se utiliza el estándar SPI para comunicar los dispositivos FPGA y *Teensy* (detallado en la sección 3.2).

3 Análisis, diseño y desarrollo

En este capítulo se describirá el proceso de análisis del problema planteado inicialmente y de las soluciones adoptadas. Se comenzará con un estudio de las necesidades de la plataforma que llevarán al planteamiento general de la solución a emplear, para posteriormente detallar las particularidades de diseño e implementación de cada una de dichas partes.

3.1. Análisis general

El objetivo del proyecto es diseñar una plataforma que permita al dispositivo VECTOR desarrollar su función como piloto automático tratando de simular las condiciones de vuelo de forma que la unidad se comporte de la misma manera que en un vuelo real.

Como se indicó en la descripción del dispositivo, presentado en la sección 2.1, los datos que precisa la UUT los recibe del dispositivo POLAR integrado, de forma que para conseguir que el piloto automático actúe como si se encontrara en vuelo se presentan dos opciones:

- Modificar el dispositivo POLAR integrado para que proporcione información de vuelo simulada.
- Extraer la unidad POLAR y simular el comportamiento de dicha unidad de forma externa.

La primera opción no es práctica. La unidad POLAR actúa de acuerdo a la información que proporcionan sus sensores integrados. Simular el comportamiento de los sensores llevaría a plantear una solución HIL para dicho dispositivo, que excede del cometido de este TFG. Se abordará, por tanto, el desarrollo de la segunda opción.

Para realizar la simulación del dispositivo POLAR se creará una aplicación de escritorio utilizando la biblioteca *Qt* que se conectará a la UUT y enviará mensajes con la misma frecuencia y formato que éste. La información de vuelo que se incluye en los paquetes se obtendrá conectando por TCP la aplicación al simulador de vuelo *X-Plane* mediante el *plugin ExtPlane*.

Para cerrar el bucle, se hace necesaria una manera de alimentar al simulador de forma apropiada con la salida producida por la UUT. Esto se llevará a cabo conectando las salidas de la unidad VECTOR a la FPGA, que las unificará en un único paquete de información. El paquete se enviará al dispositivo *Teensy*, donde la información será interpretada y traducida a

un movimiento de *joystick* que se utilizará como entrada en el simulador de vuelo. El esquema general de la plataforma HIL planteada puede verse en la Figura 3.1.

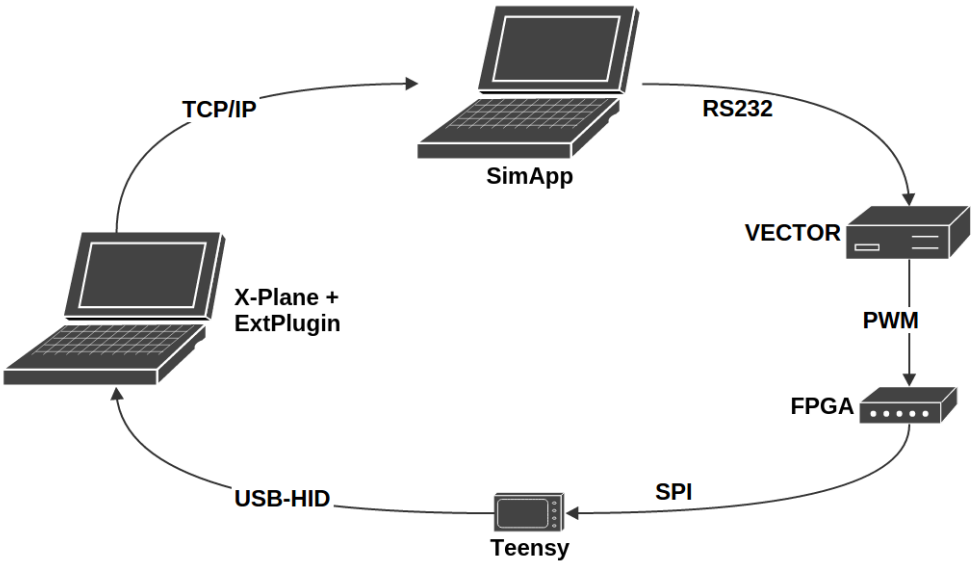


Figura 3.1: Esquema de la plataforma HIL

3.2. Fase VECTOR → *X-Plane*

En esta sección se detallará el proceso para llevar la información desde la FPGA hasta el simulador *X-Plane*. Como se indicó en la sección 2.3, no se discutirán los procesos llevados a cabo en dicho dispositivo y se analizará el proceso desde la salida producida por éste.

3.2.1. Recepción de datos en *Teensy*

La conexión entre la FPGA y la *Teensy* se lleva a cabo por SPI con los parámetros de configuración CPOL=1, CPHA=1 y DORD=0 según el esquema de la Figura 3.2 (ver anexos A y B para la correspondencia de pines). El dispositivo *Teensy* actúa como *maestro* y la FPGA como *esclavo*. Además de dicho paquete se transmite un *byte* adicional de CRC para comprobar la integridad de la información.

El formato del paquete enviado desde la FPGA se puede ver en la Figura 3.3 y su contenido está descrito en la Tabla 3.1.

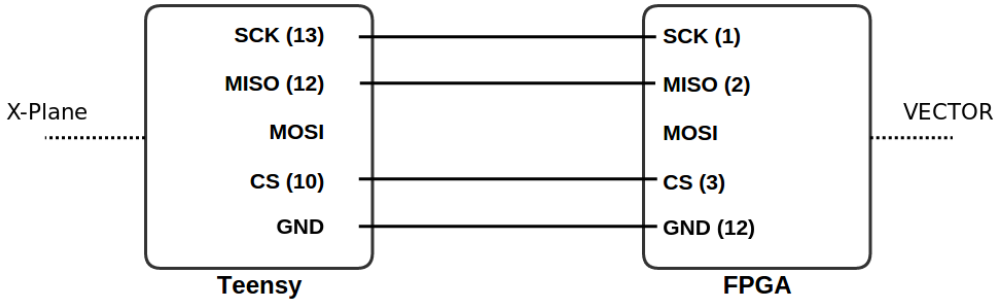


Figura 3.2: Conexión SPI entre la FPGA y la *Teensy*

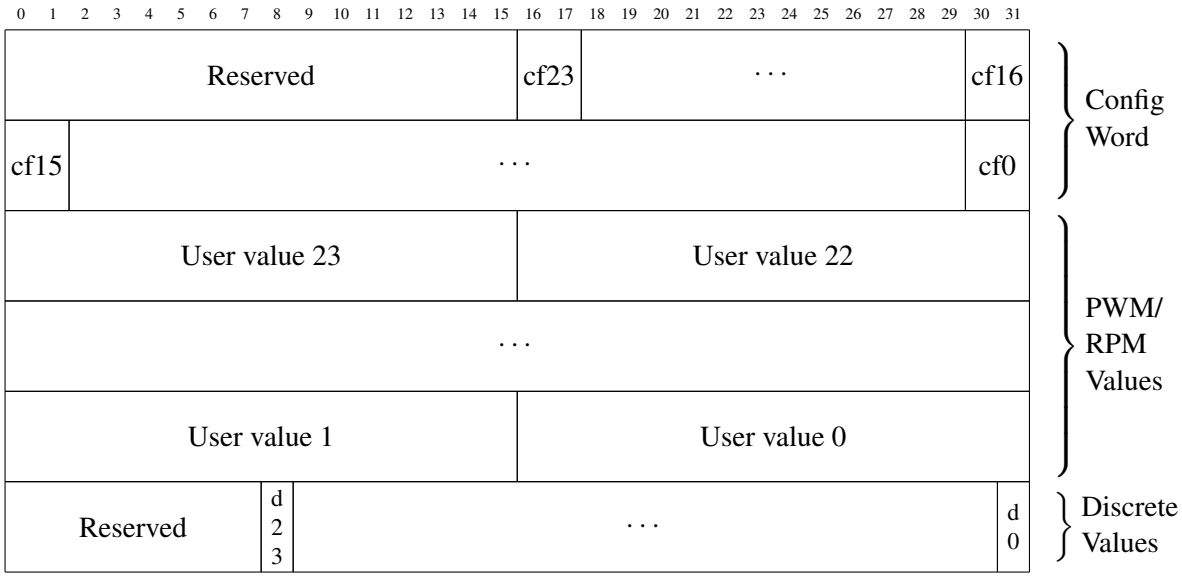


Figura 3.3: Paquete de datos transmitido entre la FPGA y la *Teensy*

En la siguiente tabla, el valor X representa un número entre 0 y 23.

Campo	Descripción
cf X	Valor de 2 *bits* que indica la configuración del campo X ('User value X' y dX), puede tener los siguientes valores: • 0 (00): El elemento X es un valor RPM. • 1 (01): El elemento X es un valor PWM. • 2 (10): El elemento X es un valor discreto. • 3 (11): No usado.
User Value X	Valor PWM ó RPM asociado al elemento X, depende del valor 'cf X': • 'cf X' = 0. Puede tomar valores entre 0 y 65536. • 'cf X' = 1. Puede tomar valores entre 800 y 2200. Además, el *bit* más significativo indica si la señal PWM es válida (valor 1) o no (valor 0). • 'cf X' = 2. Este valor debe ignorarse.
d X	Valor discreto asociado al elemento X, si 'cf X' es 0 ó 1, este campo debe ignorarse.

Tabla 3.1: Campos del paquete SPI

3.2.2. Procesamiento en *Teensy*

El objeto del dispositivo *Teensy* es simular una interfaz HID en el computador en el que se ejecuta *X-Plane*. Concretamente se simulará un dispositivo de tipo *joystick* con ejes y botones. Posteriormente, se traducirá la información recibida en el paquete SPI (Figura 3.3) a movimientos en los ejes y pulsaciones de botones en dicho *joystick*.

El requisito inicial consistía en simular un *joystick* con al menos 24 ejes y 24 botones, de forma que cada señal *User value* se asocie a un eje y cada valor discreto a un botón. El problema de este enfoque reside en que los *drivers* genéricos para *joystick*, a menudo dan problemas de compatibilidad con dispositivos de más de ocho ejes. Puesto que uno de los objetivos del proyecto es que sea fácilmente portable, se decidió desestimar la idea de crear un *driver* propio y, en su lugar, hacer que el dispositivo *Teensy* simulase 3 *joysticks* de ocho ejes cada uno.

Con este objetivo, se ha modificado el *Device Descriptor* del *joystick* por defecto de *Teensy* acomodando el nuevo diseño. Para simular 3 *joysticks* se hace uso de un campo del *Device Descriptor* que permite acompañar cada paquete de información con un identificador, de forma que los paquetes se asocian al *joystick* con el identificador correspondiente. El número total de identificadores se definen en el *Device Descriptor*.

Cada *joystick* envía los ejes correspondientes al paquete SPI de acuerdo a la Tabla 3.2. Todos los valores discretos son enviados por los botones del *joystick* 1, quedando los botones correspondientes a los *joysticks* 2 y 3 libres para futuras ampliaciones. La estructura del paquete HID completo se puede ver en la Figura 3.4

User value #	***Joystick ID***
0 - 7	1
8 - 15	2
16 - 24	3

Tabla 3.2: Asignación de señales PWM/RPM a *joysticks*

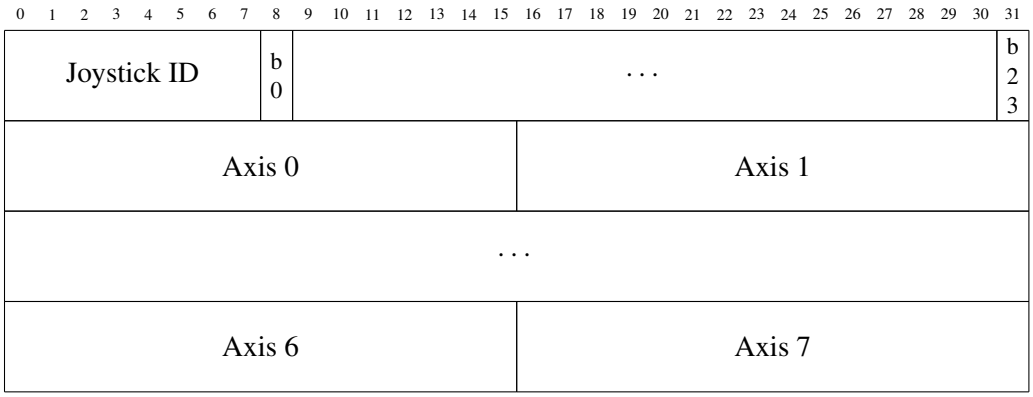


Figura 3.4: Paquete USB-HID transmitido por la *Teensy*

El bucle principal de la *Teensy* recibe un paquete SPI (Figura 3.3), comprueba que es correcto calculando su CRC, crea 3 paquetes HID correspondientes a los *joysticks* y los envía por USB.

3.2.3. Recepción en *X-Plane*

Actualmente, la asignación de los ejes y botones enviados por los 3 *joysticks* a los distintos parámetros que ofrece X-Plane, debe realizarse manualmente desde el menú de configuración de 'Joystick y Equipo'. Esta asignación es dependiente de la configuración de salida de la UUT y de la prueba que se quiera realizar.

La calibración de ejes de *joysticks* disponible en *X-Plane*, permite ajustar el rango de acción del parámetro ligado al eje al rango de valores recibidos en el *joystick* (véase la Figura 3.5). Por esta razón no es necesario hacer un escalado de los valores de tipo PWM (de 800 a 2200) al rango completo del *joystick* (de 0 a 65535). No obstante, sí es necesario un método para calibrarlos. Para ello, la FPGA dispone de un modo que ignora la información del dispositivo VECTOR y comienza a enviar paquetes barriendo completamente el rango de cada salida.

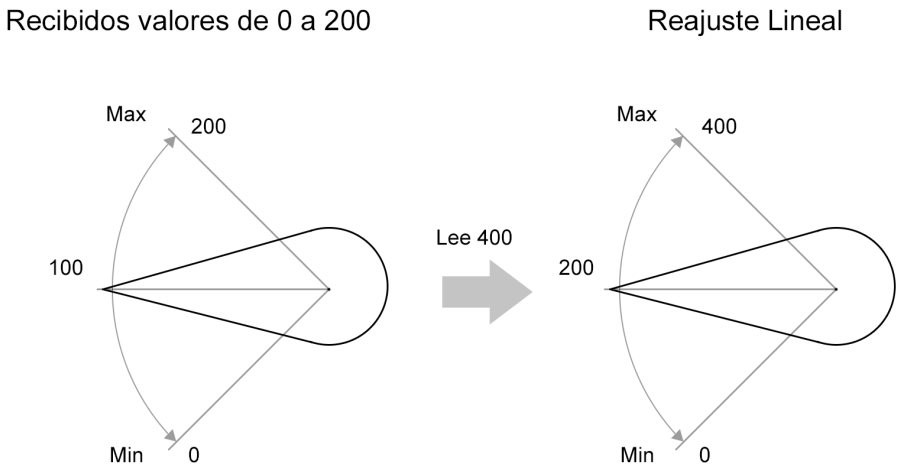


Figura 3.5: Ejemplo de reescalado automático de ejes en X-Plane

3.3. Fase *X-Plane* → VECTOR (*SimAp*)

Esta sección se centrará en la solución software ideada para simular el dispositivo POLAR y enviar datos de vuelo a la UUT. El comportamiento de la unidad POLAR se encuentra descrito en el ICD oficial [12] y en este TFG se emularán únicamente aquellos aspectos estrictamente necesarios para el dispositivo VECTOR.

3.3.1. Diseño general de la aplicación SimAp

La aplicación se ha dividido en tres módulos con funciones claramente diferenciadas y una zona de memoria compartida. En la Figura 3.6 se presenta un esquema con los módulos y la relación entre ellos. Desde el módulo que controla la interfaz gráfica, se recibe la interacción del usuario y se inicia o detiene la ejecución de los módulos *Sender* y *Updater*.

La aplicación puede encontrarse en dos estados:

- Parada (*Idle*): Estado por defecto en el momento del inicio. La aplicación se encuentra en espera, la interfaz acepta parámetros y puede pasar al estado *Running*.
- En Funcionamiento (*Running*): La aplicación se encuentra recibiendo información del simulador de vuelo *X-Plane* mediante el hilo *Updater* y enviándola a la unidad VECTOR mediante el hilo *Sender*.

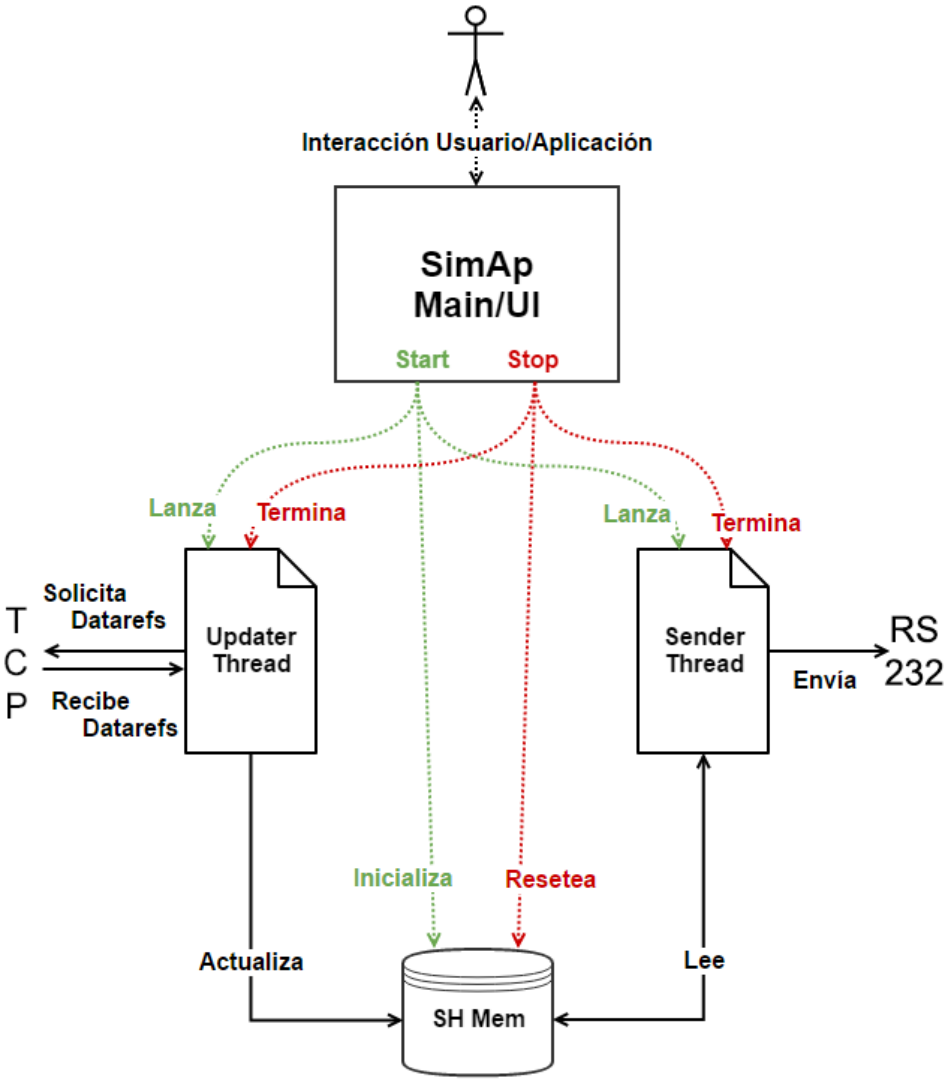


Figura 3.6: Esquema de la aplicación SimAp

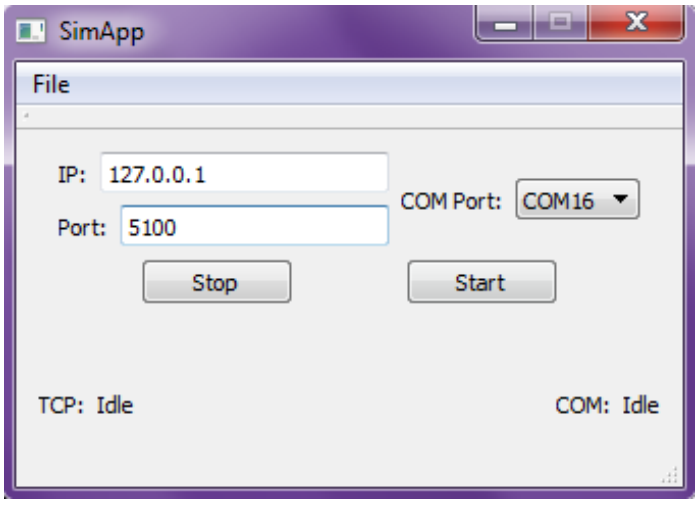


Figura 3.7: Interfaz de la aplicación SimAp

3.3.2. Interfaz gráfica

Para esta primera versión de la plataforma HIL se ha creado con *Qt* una interfaz mínima (Figura 3.7) que dispone únicamente de los elementos de configuración siguientes:

- Dirección IPv4 y puerto donde se encuentra el servidor TCP del *plugin ExtPlane*.
- Puerto Serie por el que enviar la información destinada al dispositivo VECTOR.

Además, se dispone de dos botones que inician las siguientes acciones:

- *Start*: Inicializa la zona de memoria compartida y crea los hilos *Sender* y *Updater*. Además inicializa un temporizador a 1ms y conecta su señal de *timeout* al bucle principal del hilo *Sender*. En caso de inicialización correcta, el programa pasa al estado *Running*. Cualquier error en la inicialización aborta el proceso e informa al usuario mediante un mensaje que incluye el motivo de parada. Si el programa ya se encontraba en estado *Running*, la pulsación del botón se ignora.
- *Stop*: Devuelve el programa al estado *Idle*, deteniendo la ejecución de los hilos de forma ordenada y liberando los recursos de la memoria compartida. Si el programa ya se encontraba en este estado, se ignora la pulsación.

3.3.3. *Updater*

El *Updater* es el hilo encargado de conectar con el *plugin ExtPlane* para obtener los valores de los *datarefs* necesarios para la UUT y de mantenerlos actualizados en la zona de memoria compartida.

En el momento de la creación del hilo se inicia una conexión TCP a la dirección IPv4 y puerto introducidos por el usuario en la interfaz. Si la conexión es correcta (el *plugin ExtPlane* se encuentra realmente en esa dirección), se realiza la suscripción a todos los *datarefs* (véase el Anexo C) mediante el envío de mensajes con el formato indicado en la sección 2.5. Tras una inicialización correcta, el hilo entra en su bucle principal, en el que analiza y procesa los mensajes recibidos y actualiza el valor de la variable correspondiente en memoria compartida.

Para seleccionar el parámetro *accuracy* en el momento de la suscripción, se ha tenido en cuenta el tipo y la precisión de la variable asociada que se enviará en los paquetes de información a la unidad VECTOR (detallados en la Figura 3.8), de forma que solamente se actualicen valores con una precisión relevante a fin de evitar tanto la saturación de las comunicaciones como la disminución en el rendimiento de *X-Plane*.

3.3.4. *Sender*

El *Sender* es el hilo encargado de proveer de forma constante la información de vuelo necesaria a la unidad VECTOR. Su comportamiento debe simular el del dispositivo POLAR [12], enviando la información necesaria con la temporización y formato adecuados.

El canal de comunicación se elige en la interfaz de usuario y debe ser RS232 con los parámetros de configuración: 1Mbps, 8 *bits* de datos, sin *bit* de paridad y con 1 *bit* de parada. La información se debe enviar según el protocolo TSIP presentado en la sección 2.8. El contenido de cada paquete de información se detalla en la Figura 3.8 y en la Tabla 3.3.

Campo	Uds	Tipo	Descripción
H_ID	-	-	Identificador de paquetes de telemetría de alto nivel (0xA5).
L_ID	-	-	Identificador de paquetes de telemetría de bajo nivel (0xA0).
ID_A	-	-	Identificador del paquete 1000Hz_HISPEED (0x1E).
ID_B	-	-	Identificador del paquete 200Hz_HISPEED (0x1F).
ID_C	-	-	Identificador del paquete 50Hz_HISPEED (0x20).
ID_D	-	-	Identificador del paquete System (0x05).
Eb	$rad \cdot 5e3$	short int	Ángulos de *Roll*, *Pitch* y *Yaw*.
Ab	cm/s^2	short int	Aceleración en los ejes X, Y, Z.
Ob	$mRad/s$	short int	Velocidad angular en los ejes X, Y, Z.
Vn	cm/s	short int	Velocidad en los ejes X, Y, Z.
LAT	$deg \cdot 1e7$	long	Latitud.
LON	$deg \cdot 1e7$	long	Longitud.
ALT	m	IEEE-754 single precission	Altitud.
IAS	cm/s	short int	*Indicated airspeed*.
TAS	cm/s	short int	*True airspeed*.
SN	-	short int	Número de serie de la unidad POLAR. Fijo a valor 1.
V	mV	short int	Tensión de las baterías. Valores fijos. V[0]= 12000, V[1] = 5000.
Unused	-	-	Información no utilizada. Valor 0.

Tabla 3.3: Campos de los paquetes TSIP enviados por *Sender*

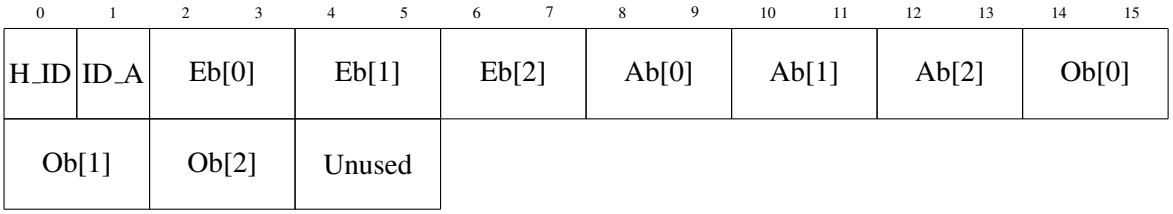
Cada paquete de información debe enviarse con una frecuencia específica (detallada en la Tabla 3.4). De cara a la implementación en *Qt* se ha creado un temporizador que emite una señal cada milisegundo y se ha conectado al *slot* principal del hilo. En ese *slot* se comprueban los paquetes que deben enviarse en ese milisegundo, se construyen a partir de los datos disponibles en memoria compartida y se envían. Cada paquete se envía con un desfase absoluto respecto al primer *tick* del temporizador. Esto evita que el envío de varios paquetes coincida en el tiempo, lo que podría provocar que se ignoren señales del temporizador por estar ya ejecutándose el *slot* asociado.

Nombre	Frecuencia (Hz)	Período (ms)
1000Hz_HISPEED	200	5
200Hz_HISPEED	200	5
50Hz_HISPEED	50	20
System	5	200

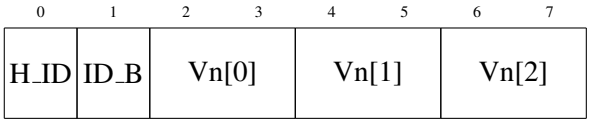
Tabla 3.4: Frecuencia de envío de cada paquete

En los siguientes paquetes, cada campo representa un *byte* y el nombre indicado se corresponde con el nombre oficial encontrado en [12].

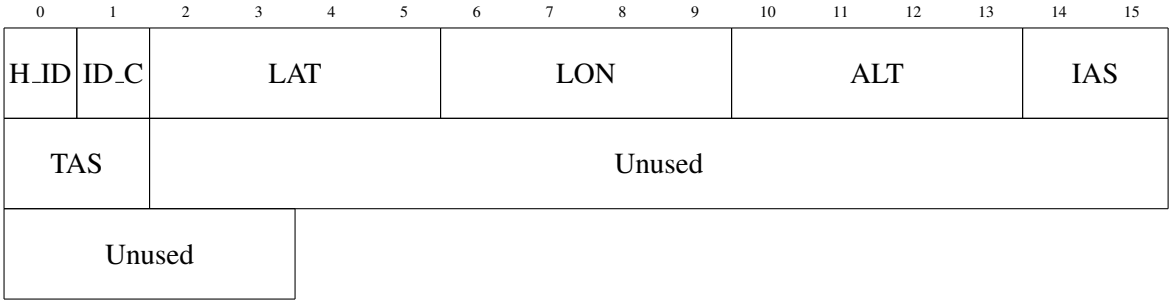
Paquete 1000Hz_HISPEED:



Paquete 200Hz_HISPEED:



Paquete 50Hz_HISPEED:



Paquete System:

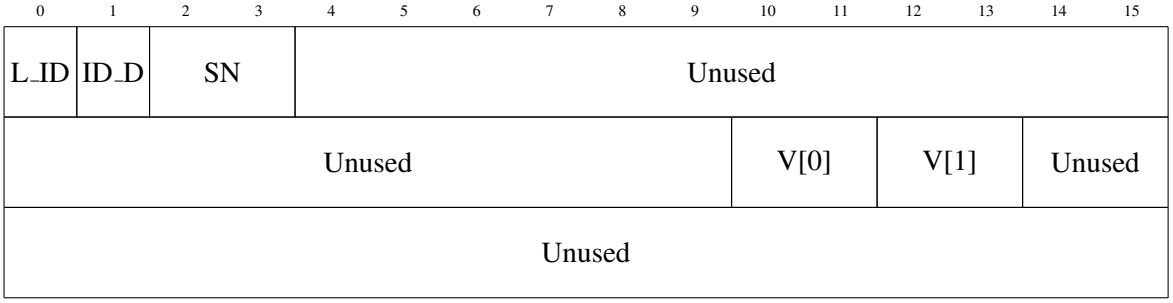


Figura 3.8: Formato de los paquetes enviados por *Sender*

3.3.5. Memoria Compartida (*SH Mem*)

Debido a la concurrencia entre los hilos *Sender* y *Updater*, se hace necesario un mecanismo de acceso a la memoria que garantice la integridad de los datos. La naturaleza periódica de las funciones que realiza el hilo *Sender* y la necesidad de mantener cierta precisión en su temporización, descartan el uso de mecanismos de exclusión mutua ante la posibilidad de bloqueos que retrasen la ejecución de dicho hilo. La solución adoptada finalmente utiliza un mecanismo libre de bloqueos basado en el uso de variables atómicas.

La implementación de la solución se ha realizado mediante una tabla *hash* que asocia la cadena de caracteres identificativa del *dataref* a la variable atómica correspondiente, esto minimiza el tiempo de acceso y permite un rendimiento óptimo del hilo *Sender*.

El formato de los datos proporcionado por *X-Plane*, en ocasiones difiere del formato en el que el dispositivo VECTOR espera recibir la información (diferentes unidades o sistemas de referencia). Este problema se ha solucionado definiendo métodos y factores para realizar la conversión entre ambos formatos.

Por tanto, la zona de memoria compartida ha sido implementada como una clase estática que agrupa la tabla hash de variables, los métodos y los factores de conversión descritos anteriormente.

4

Pruebas y resultados

La naturaleza altamente modular de la plataforma HIL, hace necesaria la validación de las funcionalidades de cada componente antes de su integración en el bucle. En este capítulo se presentarán las pruebas realizadas a cada módulo y los resultados correspondientes.

4.1. Pruebas al dispositivo *Teensy*

A continuación se presentan las pruebas realizadas al dispositivo *Teensy* y los resultados obtenidos:

- Comunicación SPI: Se creó un paquete de prueba del tipo visto en la Figura 3.3 con contenido estático para transmitir desde la FPGA y se comprobó su correcta recepción en la *Teensy* comparando los valores recibidos con el contenido original, tanto para el contenido del paquete como para el código CRC. Esta prueba se utilizó posteriormente para determinar experimentalmente la velocidad máxima de transmisión soportada por esta configuración concreta, alcanzándose hasta los 6 MHz (6 Mbps) con éxito.
- USB *Device Descriptor*: Se ha comprobado que el dispositivo se identifica correctamente como 3 *joysticks* en el computador al que se conecta la *Teensy* por USB. También se ha verificado el correcto funcionamiento de cada *joystick* mediante un programa de prueba que realiza el barrido del rango de cada eje y la pulsación intermitente de cada botón para cada uno de los *joysticks*. Actualmente el comportamiento del dispositivo en el sistema operativo *Windows 7* es el esperado. No obstante en sistemas *Linux* existen algunos problemas de compatibilidad pendientes de resolución.
- Traducción SPI-*joystick*: Se ha comprobado la corrección en las funciones que transforman paquetes SPI a paquetes de *joystick* mediante la creación de una serie de casos de prueba que incluyen distintas configuraciones para el paquete SPI y su monitorización tanto en el computador (valores interpretados por los *joysticks*), como en la *Teensy* (formato y contenido de los paquetes de *joystick* enviados).

4.2. Pruebas a *ExtPlane* y SimAp

A continuación se presentan las pruebas realizadas con el *plugin ExtPlane* y a la aplicación *SimAp*:

- Pruebas de verificación del plugin: Mediante una conexión *telnet* a *ExtPlane* se verificó el funcionamiento del mismo y el formato de los mensajes necesarios para la suscripción a *datarefs* y las correspondientes respuestas del servidor.
- Conexión al plugin desde *Qt*: Similares a las pruebas anteriores pero realizadas desde el hilo *Updater*, haciendo uso de las clases implementadas por *Qt*.
- Cambios de sistema de referencia: Se verificó la corrección de los métodos empleados para realizar los cambios de sistemas de referencia entre el empleado por *X-Plane* y el empleado por la UUT. Para ello se realizó una monitorización de las variables en la aplicación *SimAp* mientras se ejecutaba un vuelo manual en *X-Plane*.
- Interfaz, inicio y parada: Se han llevado a cabo pruebas con las siguientes condiciones de error: dirección IPv4 o puerto incorrecto, *ExtPlane* no iniciado, puerto Serie no seleccionado. Se ha comprobado que la aplicación detecta y maneja correctamente estos errores. También se han probado parámetros válidos para verificar la inicialización correcta de los hilos y la memoria compartida. Además, se ha probado la función de parada para comprobar que todos los recursos se detienen y liberan ordenadamente.
- Temporización: Se ha medido la precisión del temporizador de 1 ms utilizado por el hilo *Sender* en la plataforma *Windows 7* haciendo uso de la clase *PerformanceCounter*. Se han realizado varias mediciones de 10 minutos cada una y se ha obtenido una media exacta de 1 ms y una desviación típica desde 50μs en la mejor medición hasta 300 μs en la peor. En la Figura 4.1 se presenta una gráfica que resume los resultados obtenidos. Se considera una aproximación suficientemente aceptable para las necesidades del dispositivo VECTOR. El mejor resultado se ha obtenido asignando una prioridad de 'tiempo real' a la aplicación en el 'Administrador de Tareas' de *Windows*.

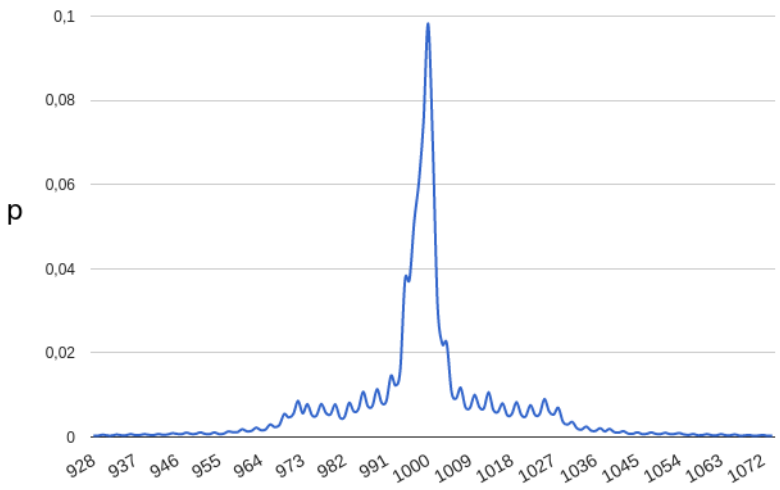


Figura 4.1: Probabilidad p de que el temporizador se active en $t = x$ μs

5 Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones obtenidas tras la realización del proyecto así como una serie de posibles líneas de trabajo futuro destinadas a mejorar y ampliar las capacidades de la plataforma HIL desarrollada.

5.1. Conclusiones

A lo largo de este TFG se ha construido una plataforma HIL para la realización de pruebas al software de piloto automático de los dispositivos de control de vuelo VECTOR. El estudio del comportamiento y necesidades de dicho dispositivo han permitido diseñar la arquitectura de la plataforma y su posterior implementación. Si bien a fecha de entrega de este documento no ha sido posible realizar pruebas de sistema para validar completamente la solución planteada, sí han sido validados todos los módulos por separado y las conexiones entre ellos. Además los ingenieros aeronáuticos que diseñan e implementan el piloto automático y a quienes va destinada, en última instancia, esta plataforma, han valorado positivamente el proyecto en su estado actual.

Los elementos que integran la plataforma se han diseñado con el objetivo secundario de ser fácilmente portables y extensibles, para lo que se han utilizado estándares como HID, RS232, SPI y la biblioteca multiplataforma *Qt*. No obstante, se han encontrado ciertas limitaciones con este enfoque. La librería *Qt* no provee temporizadores con resoluciones de orden inferior a milisegundos. En caso de querer ampliar la funcionalidad de la aplicación principal con tareas que requieran una temporización del orden de microsegundos o un ajuste más preciso que el obtenido en las pruebas de la sección 4.2, se deberá implementar una solución propia que podría llevar incluso a la utilización de sistemas operativos en tiempo real. Este hecho, no obstante, limitaría seriamente las posibilidades de portabilidad del proyecto.

5.2. Trabajo Futuro

La amplitud de la plataforma HIL plantea múltiples líneas de trabajo futuro orientadas tanto a mejorar la funcionalidad principal de la plataforma (simulación del dispositivo POLAR), como a añadir nuevas características. Algunas de ellas se listan a continuación:

- Simulación de los datos: En lugar de obtener los datos de vuelo directamente del simulador, se puede crear una capa intermedia que obtenga datos mas básicos de los sensores y aplique el mismo algoritmo que emplea el dispositivo POLAR para estimar su posición y orientación. Esto añadiría realismo respecto al comportamiento en una situación real.
- Caracterización realista de los sensores: El simulador *X-Plane* proporciona datos de vuelo con una precisión absoluta. En una situación real, los sensores utilizados para la estimación de las distintas variables se encuentran afectados por ruido que repercute en la precisión de los datos. Se plantea la creación de una capa en el bucle que simule ruido para alterar de forma realista el valor de los datos.
- Desarrollo de un plugin para *X-Plane*: El desarrollo de un plugin propio permitiría adaptar su comportamiento a las necesidades específicas del proyecto. De esta forma, podrían incluirse en él los cambios de referencia en los sistemas de coordenadas y los factores de conversión de unidades que ahora se realizan en la aplicación *SimAp*.
- Funcionalidades *SimAp*: Añadir funcionalidades a la aplicación principal, como permitir modificar manualmente la información a enviar o simular el fallo de un sensor determinado. Esta característica podría utilizarse para probar la tolerancia del piloto ante posibles fallos en el POLAR.
- Asegurar la compatibilidad del dispositivo *Teensy* y la aplicación *SimAp* con sistemas *Linux* y *OS X*.

Referencias

[1] Lista completa de Datarefs de X-Plane. http://www.xsquawkbox.net/xpsdk/docs/DataRefs.txt (Último acceso 15/06/2016).

[2] Lista de códigos de Clase HID. http://www.usb.org/developers/defined_class/ (Último acceso 18/06/2016).

[3] Terasic DE0-Nano. http://de0-nano.terasic.com.tw/ (Último acceso 18/06/2016).

[4] Device Class Deffinition for Human Interface Devices (HID) Version 1.11. http://www.usb.org/developers/hidpage/HID1_11.pdf (Último acceso 18/06/2016).

[5] ExtPlane. X-Plane Plugin for remote Dataref access. https://github.com/vranki/ExtPlane (Último acceso 22/06/2016).

[6] Qt Framework. https://www.qt.io/(Último acceso 25/06/2016).

[7] UAV Navigation. http://www.uavnavigation.com/ (Último acceso 25/06/2016).

[8] Free On-Line Dictionary of Computing. http://foldoc.org/ (Último acceso 20/06/2016).

[9] Teensy Project. https://www.pjrc.com/teensy/index.html (Último acceso 25/06/2016).

[10] X-Plane Ultra realistic flight simulator. http://www.x-plane.com/ (Último acceso 21/06/2016).

[11] Spi4Teensy Repository. https://github.com/xxxajk/spi4teensy3 (Último acceso 16/06/2016).

[12] UAV Navigation S.L. Polar UAV Navigation PICD.

[13] Normativa sobre UAVs en España. Gobierno de España. http://www.seguridadaerea.gob.es/lang_castellano/cias_empresas/trabajos/rpas/default.aspx (Último acceso 25/06/2016).

[14] Arduino Website. https://www.arduino.cc/ (Último acceso 22/06/2016).

I

Anexos

A

Tarjeta de referencia Teensy

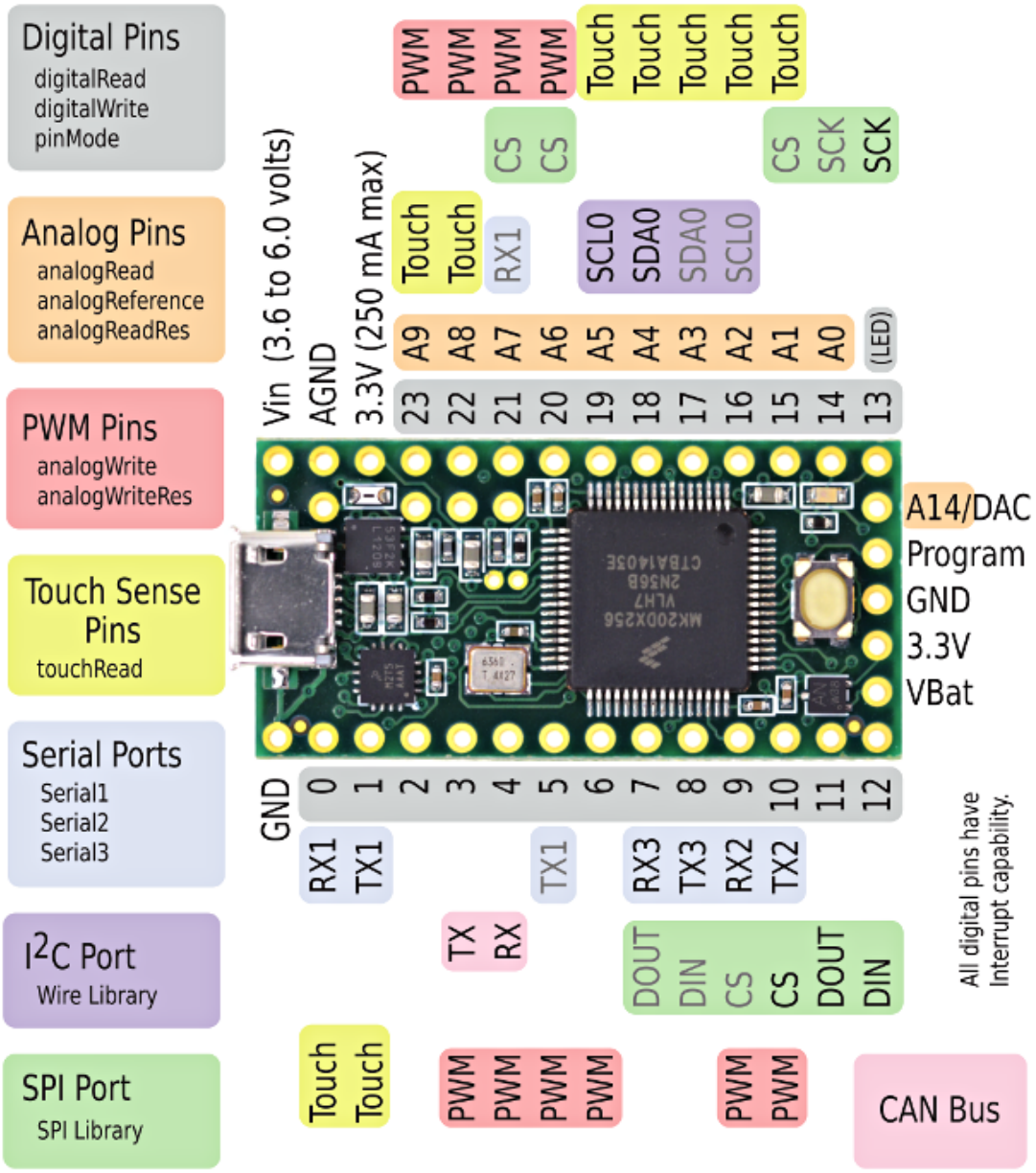


Figura A.1: Tarjeta de referencia Teensy frontal

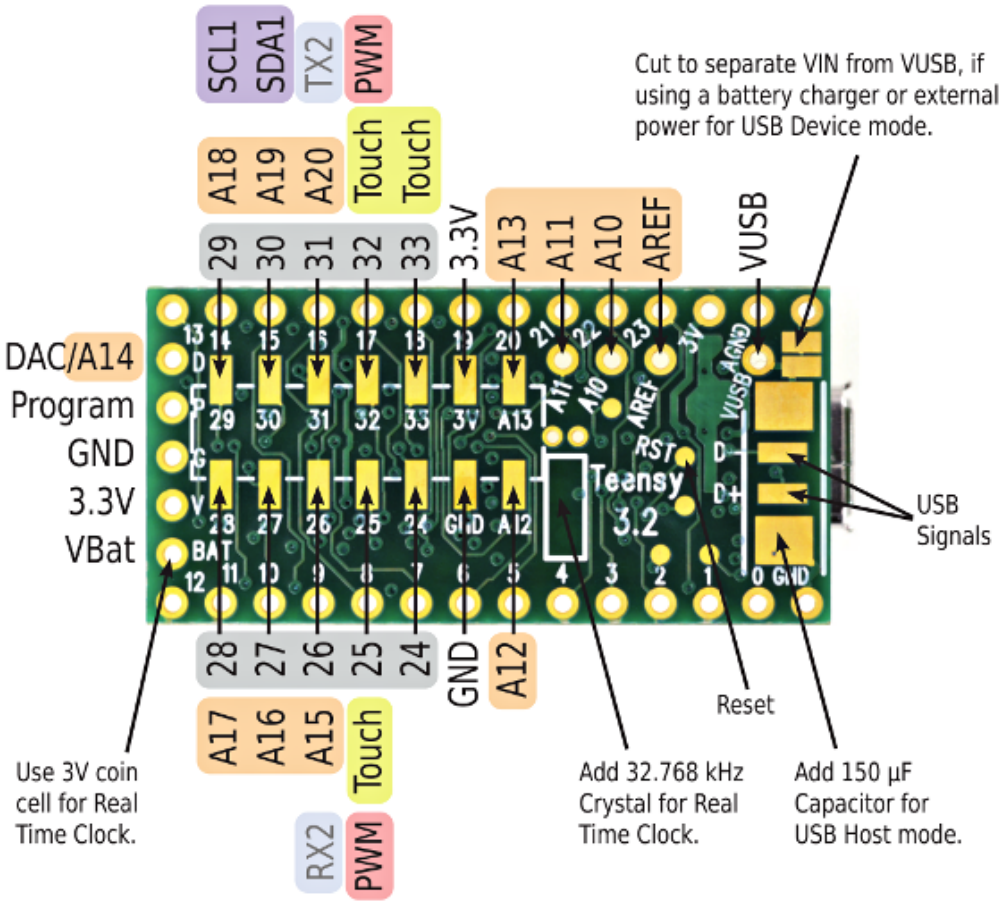
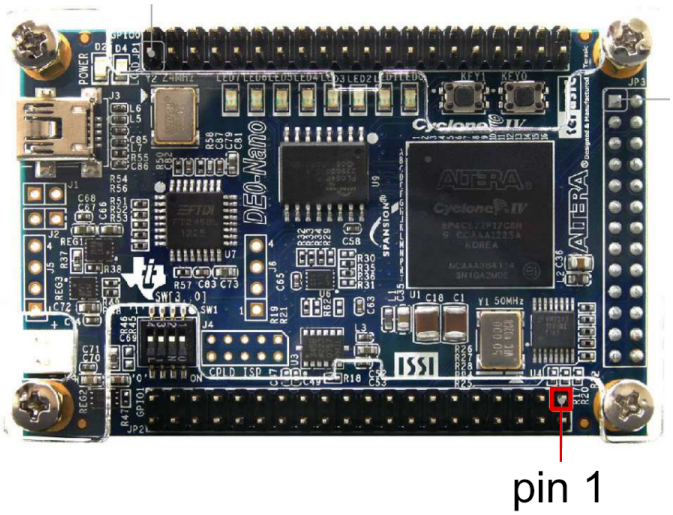


Figura A.2: Tarjeta de referencia Teensy trasera

B

Tarjeta de referencia De0-Nano



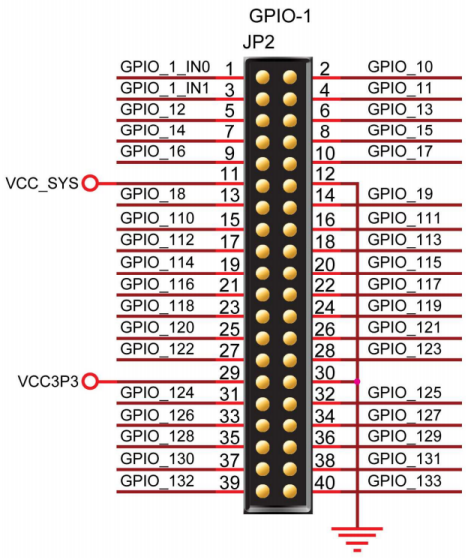


Figura B.1: Tarjeta de referencia De0-Nano

C

Correspondencia entre Datarref y variables del Polar

Variable	Dataref
Eb[0]	sim/flightmodel/position/phi
Eb[1]	sim/flightmodel/position/theta
Eb[2]	sim/flightmodel/position/psi
Ab[0]	sim/flightmodel/position/local_ax
Ab[1]	sim/flightmodel/position/local_ay
Ab[2]	sim/flightmodel/position/local_az
Ob[0]	sim/flightmodel/position/Prad
Ob[1]	sim/flightmodel/position/Qrad
Ob[2]	sim/flightmodel/position/Rrad
Vn[0]	sim/flightmodel/position/local_vx
Vn[1]	sim/flightmodel/position/local_vy
Vn[2]	sim/flightmodel/position/local_vz
LAT	sim/flightmodel/position/latitude
LON	sim/flightmodel/position/longitude
ALT	sim/flightmodel/position/elevation
IAS	sim/flightmodel/position/indicated_airspeed
TAS	sim/flightmodel/position/true_airspeed

Tabla C.1: Variables del POLAR y *datarefs* asociados